Appendix A

Software Modules

Module Title	Number of Pages
About	1 page
ImageDisplay	3 pages
RefDisplay	4 pages
SampleLocator	4 pages
Sdimain	8 pages
ShadeData	8 pages
SplashScreen	3 pages
ToothObject	6 pages

```
unit About;

interface

uses Windows, Classes, Graphics, Forms, Controls, StdCtrls,
  Buttons, ExtCtrls;

type
  TAboutBox = class(TForm)
    Panel1: TPanel;
    OKButton: TButton;
    ProgramIcon: TImage;
    ProductName: TLabel;
    Version: TLabel;
    Copyright: TLabel;
    Comments: TLabel;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  AboutBox: TAboutBox;

implementation

{$R *.DFM}

end.
```

```
unit ImageDisplay;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  TMultiP, ExtCtrls;

type
  TfrmImageDisplay = class(TForm)
    pnlImage: TPanel;
    pmiImage: TPMultiImage;
    procedure FormCreate(Sender: TObject);
    procedure pmiImagePaint(Sender: TObject);
  private
    { Private declarations }
    RefA : TRect;
    RefR : integer;
    RefC : integer;
    SampleA : TRect;
    SampleR : integer;
    SampleC : integer;
    DisplayGrid : boolean;
    procedure DrawGrid(Area : TRect; Rows, Columns : integer);
    procedure DrawGrids;
  public
    { Public declarations }
    procedure DefineGrids(RefArea: TRect;
                          RefRows : integer;
                          RefCols : integer;
                          SampleArea : TRect;
                          SampleRows : integer;
                          SampleCols : integer);

    procedure HideGrid;
  end;

var
  frmImageDisplay: TfrmImageDisplay;

implementation

{$R *.DFM}

procedure TfrmImageDisplay.DrawGrid(Area : TRect; Rows, Columns : integer);
var
  Spacing : real;
  index : integer;
  ScaleX : real;
  ScaleY : real;
  Left, Right, Top, Bottom : integer;
begin
  ScaleX := pmiImage.Width/640;
  ScaleY := pmiImage.Height/480;
  Left := Round(Area.Left * ScaleX);
  Right := Round(Area.Right * ScaleX);
  Top := Round(Area.Top * ScaleY);
  Bottom := Round(Area.Bottom * ScaleY);
```

```
with pmiImage.Canvas do
begin
 : moveto(Left, Top);
  lineto(Left, Bottom);
  lineto(Right, Bottom);
  lineto(Right, Top);
  lineto(Left, Top);

  if Rows > 1 then
  begin
    Spacing := (Bottom - Top) / Rows;
    for index := 1 to Rows - 1 do
    begin
      moveto(Left+1, Top + trunc(Spacing * index));
      lineto(Right-1, Top + trunc(Spacing * index));
    end;
  end;

  if Columns > 1 then
  begin
    Spacing := (Right - left) / Columns;
    for index := 1 to Columns - 1 do
    begin
      moveto(Left + trunc(Spacing * index), Top+1);
      lineto(Left + trunc(Spacing * index), Bottom-1);
    end;
  end;
 end;
end; // DrawGrid

procedure TfrmImageDisplay.DrawGrids;
begin
    SetROP2(pmiImage.Canvas.Handle, R2_NOT);

  if DisplayGrid then
  begin
    DrawGrid(RefA, RefR, RefC);
    DrawGrid(SampleA, SampleR, SampleC);
  end;
end; // DrawGrids


procedure TfrmImageDisplay.FormCreate(Sender: TObject);
begin
  DisplayGrid := false;
end;

procedure TfrmImageDisplay.DefineGrids(RefArea: TRect;
                                       RefRows : integer;
                                       RefCols : integer;
                                       SampleArea : TRect;
                                       SampleRows : integer;
                                       SampleCols : integer);

begin
  RefA := RefArea;
  RefR := RefRows;
```

```
    RefC := RefCols;
    SampleA := SampleArea;
    SampleR := SampleRows;
    SampleC := SampleCols;
    DisplayGrid := true;
    pmiImage.Repaint;
  end;

  procedure TfrmImageDisplay.HideGrid;
  begin
    DisplayGrid := false;
    Repaint;
  end;

  procedure TfrmImageDisplay.pmiImagePaint(Sender: TObject);
  begin
    DrawGrids;
  end;

  end.
```

```
unit RefDisplay;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Grids, ShadeData, ComCtrls, StdCtrls, ExtCtrls;

type
  TfrmReferenceDisplay = class(TForm)
    Panel1: TPanel;
    sgAnalysis: TStringGrid;
    Panel2: TPanel;
    edGridCol: TEdit;
    udGridCol: TUpDown;
    edGridRow: TEdit;
    udGridRow: TUpDown;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure InsertShadeData(Shade : TShadeColours);
    procedure edGridColChange(Sender: TObject);
    procedure LoadShades(Shades : TShadeReferences);
    procedure edGridRowChange(Sender: TObject);
  private
    { Private declarations }
    RowInsertIndex : integer;
    DisplayRow : integer;
    DisplayColumn : integer;
    DisplayShades : TShadeReferences;
    procedure ShowShades;
  public
    { Public declarations }
  end;

var
  frmReferenceDisplay: TfrmReferenceDisplay;

implementation

{$R *.DFM}

const
  TitleRow = 0;
  NameColumn = 0;
  RedColumn = NameColumn + 1;
  GreenColumn = RedColumn + 1;
  BlueColumn = GreenColumn + 1;
  VariationColumn = BlueColumn + 1;
  RedMaxColumn = VariationColumn + 1;
  RedMinColumn = RedMaxColumn + 1;
  GreenMaxColumn = RedMinColumn + 1;
  GreenMinColumn = GreenMaxColumn + 1;
  BlueMaxColumn = GreenMinColumn + 1;
  BlueMinColumn = BlueMaxColumn + 1;
```

```
  Precision = 6;
  Digits = 4;


procedure TfrmReferenceDisplay.FormCreate(Sender: TObject);
begin
  { Configure the Grid to Display the Reference Set }
  DisplayColumn := GridWidth div 2;
  DisplayRow := GridHeight div 2;

  udGridCol.Min := 1;
  udGridCol.Max := GridWidth;

  udGridRow.Min := 1;
  udGridRow.Max := GridHeight;

  udGridRow.Position := DisplayRow;
  udGridCol.Position := DisplayColumn;

  edGridCol.Text := IntToStr(DisplayColumn);
  edGridRow.Text := IntToStr(DisplayRow);

  sgAnalysis.ColCount := 11;
  sgAnalysis.RowCount := 17; // may change

  RowInsertIndex := TitleRow + 1;
  sgAnalysis.Cells[NameColumn, TitleRow] := 'Shade';
  sgAnalysis.Cells[RedColumn, TitleRow] := 'Red';
  sgAnalysis.Cells[GreenColumn, TitleRow] := 'Green';
  sgAnalysis.Cells[BlueColumn, TitleRow] := 'Blue';
  sgAnalysis.Cells[VariationColumn, TitleRow] := 'Variation';
  sgAnalysis.Cells[RedMaxColumn, TitleRow] := 'Max Red';
  sgAnalysis.Cells[GreenMaxColumn, TitleRow] := 'Max Green';
  sgAnalysis.Cells[BlueMaxColumn, TitleRow] := 'Max Blue';
  sgAnalysis.Cells[RedMinColumn, TitleRow] := 'Min Red';
  sgAnalysis.Cells[GreenMinColumn, TitleRow] := 'Min Green';
  sgAnalysis.Cells[BlueMinColumn, TitleRow] := 'Min Blue';

  DisplayShades := TShadeReferences.Create;
end;

procedure TfrmReferenceDisplay.InsertShadeData(Shade : TShadeColours);
var
  Variation : real;
begin
  with Shade.GridColours[DisplayColumn,DisplayRow] do
  begin
    sgAnalysis.Cells[NameColumn, RowInsertIndex] := Shade.Name;
    sgAnalysis.Cells[RedColumn, RowInsertIndex] := FloatToStrF(Red, ffFixed,
Precision, Digits);
    sgAnalysis.Cells[GreenColumn, RowInsertIndex] := FloatToStrF(Green, ffFixed,
Precision, Digits);
    sgAnalysis.Cells[BlueColumn, RowInsertIndex] := FloatToStrF(Blue, ffFixed,
Precision, Digits);
    sgAnalysis.Cells[RedMaxColumn, RowInsertIndex] := FloatToStrF(RedMax,
ffFixed, Precision, Digits);
```

```
    sgAnalysis.Cells[GreenMaxColumn, RowInsertIndex] := FloatToStrF(GreenMax,
ffFixed, Precision, Digits);
    sgAnalysis.Cells[BlueMaxColumn, RowInsertIndex] := FloatToStrF(BlueMax,
ffFixed, Precision, Digits);
    sgAnalysis.Cells[RedMinColumn, RowInsertIndex] := FloatToStrF(RedMin,
ffFixed, Precision, Digits);
    sgAnalysis.Cells[GreenMinColumn, RowInsertIndex] := FloatToStrF(GreenMin,
ffFixed, Precision, Digits);
    sgAnalysis.Cells[BlueMinColumn, RowInsertIndex] := FloatToStrF(BlueMin,
ffFixed, Precision, Digits);
    Variation := (RedMax - RedMin) + (GreenMax - GreenMin) + (BlueMax -
BlueMin);
    sgAnalysis.Cells[VariationColumn, RowInsertIndex] := FloatToStrF(Variation,
ffFixed, Precision, Digits);
  end;
  inc(RowInsertIndex);
end;

procedure TfrmReferenceDisplay.ShowShades;
var
  ShadeIndex : integer;
  CurrentShade : TShadeColours;
begin
  RowInsertIndex := TitleRow + 1;
  for ShadeIndex := 0 to DisplayShades.ShadeList.Count - 1 do
  begin
    CurrentShade := DisplayShades.ShadeList.Items[ShadeIndex];
    InsertShadeData(CurrentShade);
  end;
end;

procedure TfrmReferenceDisplay.LoadShades(Shades : TShadeReferences);
var
  lDisplayRow : integer;
begin
  { First Clear Old list }
  if DisplayShades.ShadeList.Count > 0 then
    for lDisplayRow := 1 to DisplayShades.ShadeList.Count do
      sgAnalysis.Rows[lDisplayRow].Clear;

  //DisplayShades.Free;

  DisplayShades := Shades;
  sgAnalysis.RowCount := Shades.ShadeList.Count + 1;
  ShowShades;
end;

procedure TfrmReferenceDisplay.edGridColChange(Sender: TObject);
begin
  if Visible then
  begin
    DisplayColumn := StrToInt(edGridCol.Text);
    ShowShades;
  end;
end;

procedure TfrmReferenceDisplay.edGridRowChange(Sender: TObject);
```

```
begin
  if Visible then
  begin
    DisplayRow := StrToInt(edGridRow.Text);
    ShowShades;
  end;
end;

end.
```

```
unit SampleLocator;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  TMultiP, ExtCtrls, StdCtrls;

type
  TfrmSampleLocator = class(TForm)
    Panel1: TPanel;
    pmiImage: TPMultiImage;
    OpenDialog: TOpenDialog;
    btnLoadSample: TButton;
    edXPos: TEdit;
    Label1: TLabel;
    edYPos: TEdit;
    Label2: TLabel;
    rgLocation: TRadioGroup;
    Panel2: TPanel;
    Label3: TLabel;
    Label4: TLabel;
    edRefX: TEdit;
    Label5: TLabel;
    edRefY: TEdit;
    Label6: TLabel;
    edSampleX: TEdit;
    Label7: TLabel;
    edSampleY: TEdit;
    Label8: TLabel;
    btnSave: TButton;
    btnCancel: TButton;
    procedure pmiImageMouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure FormCreate(Sender: TObject);
    procedure btnSaveClick(Sender: TObject);
    procedure btnLoadSampleClick(Sender: TObject);
    procedure pmiImageClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure btnCancelClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    ReferenceLocation : TPoint;
    SampleLocation : TPoint;


  end;

var
  frmSampleLocator: TfrmSampleLocator;

implementation

{$R *.DFM}
```

```
uses
  SDIMain, IniFiles;

procedure TfrmSampleLocator.pmiImageMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  edXPos.Text := IntToStr(X);
  edYPos.Text := IntToStr(Y);
end;

procedure TfrmSampleLocator.FormCreate(Sender: TObject);
var
  IniFile : TIniFile;
begin
  { Load The Saved Sample Location From Ini File }
  { Set Default for now }
  ReferenceLocation := Point(170, 40);
  SampleLocation := Point(300, 160);

  if FileExists(frmShadeAnalyzer.DiskDrive + 'Analyse\' + IniFileName) then
  begin
    try
      IniFile := TIniFile.Create(frmShadeAnalyzer.DiskDrive + 'Analyse\' +
IniFileName);

      with ReferenceLocation do
      begin
        X := StrToInt(IniFile.ReadString(IniReferenceSection,IniRefX,'ERROR'));
        Y := StrToInt(IniFile.ReadString(IniReferenceSection,IniRefY,'ERROR'));
      end;

      with SampleLocation do
      begin
        X := StrToInt(IniFile.ReadString(IniSampleSection,IniSampleX,'ERROR'));
        Y := StrToInt(IniFile.ReadString(IniSampleSection,IniSampleY,'ERROR'));
      end;
    finally
      IniFile.Free;
    end;
  end;
end;

procedure TfrmSampleLocator.btnSaveClick(Sender: TObject);
var
  IniFile : TIniFile;
begin
  ReferenceLocation := point(StrToInt(edRefX.Text), StrToInt(edRefY.Text));
  SampleLocation := point(StrToInt(edSampleX.Text), StrToInt(edSampleY.Text));

  if FileExists(frmShadeAnalyzer.DiskDrive + 'Analyse\' + IniFileName) then
  begin
    try
      IniFile := TIniFile.Create(frmShadeAnalyzer.DiskDrive + 'Analyse\' +
IniFileName);

      with ReferenceLocation do
```

```
      begin
        IniFile.WriteString(IniReferenceSection,IniRefX,IntToStr(X));
        IniFile.WriteString(IniReferenceSection,IniRefY,IntToStr(Y));
      end;

      with SampleLocation do
      begin
        IniFile.WriteString(IniSampleSection,IniSampleX,IntToStr(X));
        IniFile.WriteString(IniSampleSection,IniSampleY,IntToStr(Y));
      end;
    finally
      IniFile.Free;
    end;
  end;

  Close;
end;

procedure TfrmSampleLocator.btnLoadSampleClick(Sender: TObject);
begin
  OpenDialog.Title := 'Sample Imade To Display';
  OpenDialog.InitialDir := Copy(ParamStr(0),0,3) + 'Analyse\Pictures\';
  OpenDialog.DefaultExt := GraphicExtension(TBitmap);
  OpenDialog.Filter := GraphicFilter(TBitmap);
  OpenDialog.Options := [ofPathMustExist, ofFileMustExist];
  if OpenDialog.Execute then
  begin
    pmiImage.Picture.LoadFromFile(OpenDialog.Filename);
  end;
end;

procedure TfrmSampleLocator.pmiImageClick(Sender: TObject);
begin
  { Check Location Option and Update The Location Co Ords }
  if rgLocation.ItemIndex = 0 then
  begin
    edRefX.Text := edXPos.Text;
    edRefY.Text := edYPos.Text;
  end
  else
  begin
    edSampleX.Text := edXPos.Text;
    edSampleY.Text := edYPos.Text;
  end;
end;

procedure TfrmSampleLocator.FormShow(Sender: TObject);
begin
  edRefX.Text := IntToStr(ReferenceLocation.x);
  edRefY.Text := IntToStr(ReferenceLocation.y);
  edSampleX.Text := IntToStr(SampleLocation.x);
  edSampleY.Text := IntToStr(SampleLocation.y);
end;

procedure TfrmSampleLocator.btnCancelClick(Sender: TObject);
begin
  Close;
```

```
end;
end.
```

```
unit Sdimain;

interface

uses Windows, Classes, Graphics, Forms, Controls, Menus,
  Dialogs, StdCtrls, Buttons, ExtCtrls, ComCtrls,
  ShadeData;

const
  IniFilename             = 'ShadeAnalyse.ini';

  IniReferenceSection     = 'REFERENCE AREA';
  IniRefX                 = 'RefAreaX';
  IniRefY                 = 'RefAreaY';

  IniSampleSection        = 'SAMPLE AREA';
  IniSampleX              = 'SampleAreaX';
  IniSampleY              = 'SampleAreaY';

  IniShadeSetSection      = 'DEFAULT GUIDE';
  IniDefaultGuide         = 'GuideFilename';

  Startup : Boolean       = true;  // used for splash screen

type
  TfrmShadeAnalyzer = class(TForm)
    SDIAppMenu: TMainMenu;
    FileMenu: TMenuItem;
    ExitItem: TMenuItem;
    N1: TMenuItem;
    OpenDialog: TOpenDialog;
    Help1: TMenuItem;
    About1: TMenuItem;
    StatusBar: TStatusBar;
    Calibrate: TMenuItem;
    Options1: TMenuItem;
    ShowImage1: TMenuItem;
    ShowReference1: TMenuItem;
    SetSampleLoc1: TMenuItem;
    Analyse1: TMenuItem;
    gbShadeSet: TGroupBox;
    btnLoad: TButton;
    btnSave: TButton;
    edShadeSetName: TEdit;
    gbSampleAnalysis: TGroupBox;
    btnMatch: TButton;
    Label1: TLabel;
    edNearest: TEdit;
    SaveDialog: TSaveDialog;
    procedure ShowHint(Sender: TObject);
    procedure About1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure CalibrateClick(Sender: TObject);
    procedure ShowImage1Click(Sender: TObject);
    procedure ShowReference1Click(Sender: TObject);
    procedure SetSampleLoc1Click(Sender: TObject);
    procedure Analyse1Click(S nder: TObject);
```

```
    procedure btnSaveClick(Sender: TObject);
    procedure btnLoadClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormActivate(Sender: TObject);
  private
    { Private declarations }
    Shades : TShadeReferences;
    function AnalyseImage(FileName : string; ShadeName : string) :
TShadeColours;
    function FindNearestShade(Sample : tShadeColours) : string;
    procedure LoadShadeSet(Filename : string);
  public
    { Public declarations }
    DiskDrive : string;
    NewCalibration : boolean;
  end;

var
  frmShadeAnalyzer: TfrmShadeAnalyzer;

implementation

uses
  SysUtils, About, IniFiles,
  ToothObject, ImageDisplay, RefDisplay, SampleLocator, SplashScreen;

{$R *.DFM}

const
  RefRedMedian = 0.5432;
  RefGreenMedian = 0.6308;
  RefBlueMedian = 0.3355;

   RefRows = 1;
   RefColumns = 1;
   SampleRows = GridHeight;     // To change see Shade Data
   SampleColumns = GridWidth;


procedure TfrmShadeAnalyzer.ShowHint(Sender: TObject);
begin
  StatusBar.SimpleText := Application.Hint;
end;

procedure TfrmShadeAnalyzer.About1Click(Sender: TObject);
begin
  AboutBox.ShowModal;
end;

procedure TfrmShadeAnalyzer.FormCreate(Sender: TObject);
var
  IniFile : TIniFile;
  DefaultShadeFilename : string;
begin
  Application.OnHint := ShowHint;
  DiskDrive := Copy(ParamStr(0),0,3);
  Shades := TShadeReferences.Create; // we will build a new list
```

```
    NewCalibration := false;
    try
      IniFile := TIniFile.Create(DiskDrive + 'Analyse\' + IniFilename);
      DefaultShadeFilename := IniFile.ReadString(IniShadeSetSection,
  IniDefaultGuide, 'ERROR');
      DefaultShadeFilename := DiskDrive + 'Analyse\' + DefaultShadeFilename;
      LoadShadeSet(DefaultShadeFilename);
    finally
      IniFile.Free;
    end;
  end;

function TfrmShadeAnalyzer.AnalyseImage(FileName : string; ShadeName : string) :
TShadeColours;
var
    ShadeColours : TShadeColours;
    Tooth : TTooth;
    DeltaRed : real;
    DeltaGreen : real;
    DeltaBlue : real;
    PixelPercent : real;
    RefArea : TRect;
    SampleArea : TRect;
  begin
    Tooth := TTooth.Create;

    { Analyse The Reference Area }
    frmImageDisplay.HideGrid;
    Tooth.LoadBitmapFromFile(FileName);
    frmImageDisplay.pmiImage.Picture.Bitmap.Assign(Tooth.ToothBitmap);
    Application.ProcessMessages;

    RefArea := Tooth.FillSearchSampleLimits(frmSampleLocator.ReferenceLocation);
    frmImageDisplay.DefineGrids(RefArea, RefRows, RefColumns, Rect(0,0,0,0),
  SampleRows, SampleColumns);
    Application.ProcessMessages;

    Tooth.RemoveMask(RefArea);
    frmImageDisplay.pmiImage.Picture.Bitmap.Assign(Tooth.ToothBitmap);
    Application.ProcessMessages;

    Tooth.Analyse(0, 0, RefArea, RefRows, RefColumns, DeltaRed, DeltaGreen,
  DeltaBlue, PixelPercent);

    DeltaRed := RefRedMedian - DeltaRed;
    DeltaGreen := RefGreenMedian - DeltaGreen;
    DeltaBlue := RefBlueMedian - DeltaBlue;

    { Now Analyse the Sample Area }
    frmImageDisplay.HideGrid;
    Tooth.LoadBitmapFromFile(FileName);
    frmImageDisplay.pmiImage.Picture.Bitmap.Assign(Tooth.ToothBitmap);
    Application.ProcessMessages;

    SampleArea := Tooth.FillSearchSampleLimits(frmSampleLocator.SampleLocation);
    frmImageDisplay.DefineGrids(Rect(0,0,0,0), RefRows, RefColumns, SampleArea,
  SampleRows, SampleColumns);
```

```
  Application.ProcessMessages;

  Tooth.RemoveReflection(SampleArea);
  frmImageDisplay.pmiImage.Picture.Bitmap.Assign(Tooth.ToothBitmap);
  Application.ProcessMessages;

  ShadeColours := Tooth.AnalyseGrid(SampleArea, SampleRows, SampleColumns,
                                    DeltaRed, DeltaGreen, DeltaBlue);


  ShadeColours.Name := ShadeName;
  Result := ShadeColours;
  Tooth.Free;
end;

procedure TfrmShadeAnalyzer.CalibrateClick(Sender: TObject);
var
  FilePath : string;
  FileIndex : integer;
  lFilename : string;
  ShadeName : string;
  ProgressBar : TProgressBar;
  ShadeColours : TShadeColours;
begin
  OpenDialog.Title := 'Files To Analyse';
  OpenDialog.InitialDir := DiskDrive + 'Analyse\Pictures\';
  OpenDialog.DefaultExt := GraphicExtension(TBitmap);
  OpenDialog.Filter := GraphicFilter(TBitmap);
  OpenDialog.Options := [ofAllowMultiSelect, ofPathMustExist, ofFileMustExist];
  if OpenDialog.Execute then
    with OpenDialog.Files do
    begin
      edShadeSetName.Text := 'New Calibration';
      NewCalibration := true;
      StatusBar.SimpleText := 'Loading Calibration Bitmaps';
      Shades.Free;
      Shades := TShadeReferences.Create; // we will build a new list

      FilePath := ExtractFilePath(OpenDialog.FileName);
      ProgressBar := TProgressBar.Create(self);
      ProgressBar.Parent := self;
      ProgressBar.Align := alBottom;
      ProgressBar.Min := 0;
      ProgressBar.Max := Count+2;
      ProgressBar.Step := 1; // the amount to move with the StepIt method
      for FileIndex := 0 to Count - 1 do
      begin
        lFileName := Strings[FileIndex];

        { Get the Shade Name from the filename }
        ShadeName := ExtractFilename(lFilename);
        StatusBar.SimpleText := 'Loading Calibration Bitmap :'+ShadeName;
        ShadeName := UpperCase(Copy(ShadeName, 1, Pos('.', ShadeName) - 2)); //
remove the letter

        ShadeColours := AnalyseImage(lFileName, ShadeName);

        Shades.AddSample(ShadeColours);
```

```
      ProgressBar.Stepit; // Move one Step amount
    end;

    Application.ProcessMessages;

    { Get the Shades into alphabetical order }
    StatusBar.SimpleText := 'Sorting Shade Samples';
    Shades.SortList;
    ProgressBar.Stepit; // Move one Step amount

    { Process the Shades Data to get average sets }
    StatusBar.SimpleText := 'Reducing Shades to Reference Set';
    Shades.ReduceList;
    ProgressBar.Stepit; // Move one Step amount

    Shades.SortList;

    ProgressBar.Free;

    StatusBar.SimpleText := 'Done';
  end;
end;

procedure TfrmShadeAnalyzer.ShowImage1Click(Sender: TObject);
begin
  frmImageDisplay.Show;
end;

function TfrmShadeAnalyzer.FindNearestShade(Sample : tShadeColours) : string;
var
  CurrentShade : TShadeColours;
  ShadeName : string;
  ShadeDifference : real;
  CurrentDifference : real;
  ShadeIndex : integer;
begin
  ShadeDifference := 1000000;
  ShadeName := 'None';
  for ShadeIndex := 0 to Shades.ShadeList.Count - 1 do
  begin
    CurrentShade := Shades.ShadeList.Items[ShadeIndex];
    CurrentDifference := CurrentShade.ColourDifference(Sample);
    if CurrentDifference < ShadeDifference then
    begin
      ShadeName := CurrentShade.Name;
      ShadeDifference := CurrentDifference;
    end;
  end;
  result := ShadeName;
end;

procedure TfrmShadeAnalyzer.ShowReference1Click(Sender: TObject);
begin
  frmReferenceDisplay.LoadShades(Shades);
  frmReferenceDisplay.ShowModal;
end;
```

```
procedure TfrmShadeAnalyzer.SetSampleLoc1Click(Sender: TObject);
begin
  frmSampleLocator.ShowModal;
end;

procedure TfrmShadeAnalyzer.Analyse1Click(Sender: TObject);
var
  lFilename : string;
  SampleColours : TShadeColours;
  ProgressBar : TProgressBar;
begin
  OpenDialog.Title := 'Files To Analyse';
  OpenDialog.InitialDir := DiskDrive + 'Analyse\Pictures\';
  OpenDialog.DefaultExt := GraphicExtension(TBitmap);
  OpenDialog.Filter := GraphicFilter(TBitmap);
  OpenDialog.Options := [ofPathMustExist, ofFileMustExist];
  if OpenDialog.Execute then
  begin
    edNearest.Text := '';
    StatusBar.SimpleText := 'Analyzing Sample';

    ProgressBar := TProgressBar.Create(self);
    ProgressBar.Parent := self;
    ProgressBar.Align := alBottom;
    ProgressBar.Min := 0;
    ProgressBar.Max := 3;
    ProgressBar.Step := 1; // the amount to move with the StepIt method

    lFilename := OpenDialog.Filename;

    ProgressBar.StepIt;
    Application.ProcessMessages;

    SampleColours := AnalyseImage(lFileName, 'Unknown');

    ProgressBar.StepIt;
    Application.ProcessMessages;

    StatusBar.SimpleText := 'Searching for Nearest Shade';
    edNearest.Text := FindNearestShade(SampleColours);

    ProgressBar.StepIt;
    Application.ProcessMessages;

    StatusBar.SimpleText := 'Done';
    ProgressBar.Free;
  end;
end;

procedure TfrmShadeAnalyzer.btnSaveClick(Sender: TObject);
var
  lFilename : string;
  OutStream : TFileStream;
  IniFile : TIniFile;
begin
  SaveDialog.Title := 'Shade Guide Filename to Save';
  SaveDialog.InitialDir := DiskDrive + 'Analyse\';
```

```
SaveDialog.DefaultExt := 'SDS';
SaveDialog.Filter := 'Shade Guides|*.SDS';
SaveDialog.Options := [ofPathMustExist];
if SaveDialog.Execute then
  with SaveDialog do
  begin
    if not FileExists(Filename) or
       (MessageDlg(Format('Overwrite %s?', [ExtractFilename(Filename)]),
                   mtConfirmation, [mbYes, mbNo], 0) = mrYes) then

    begin
      try
        OutStream := TFileStream.Create(Filename, fmCreate or fmShareCompat);
        Shades.SaveToStream(OutStream);
        lFilename := ExtractFilename(SaveDialog.Filename);
        edShadeSetName.Text := copy(lFilename,1, Length(lFilename) - 4);
        NewCalibration := false;
        try
          IniFile := TIniFile.Create(DiskDrive + 'Analyse\' + IniFilename);
          IniFile.WriteString(IniShadeSetSection, IniDefaultGuide, lFilename);
        finally
          IniFile.Free;
        end;
      finally
        OutStream.Free;
      end;
    end;
  end;
end;

procedure TfrmShadeAnalyzer.LoadShadeSet(Filename : string);
var
  InStream : TFileStream;
  IniFile : TIniFile;
  lFilename : string;
begin
  try
    edShadeSetName.Text := 'Loading...';
    InStream := TFileStream.Create(Filename, fmOpenRead or fmShareCompat);
    Shades.Free;
    Shades := TShadeReferences.Create; // we will build a new list
    Shades.LoadFromStream(InStream);
    lFilename := ExtractFilename(Filename);
    edShadeSetName.Text := copy(lFilename,1, Length(lFilename) - 4);
    try
      IniFile := TIniFile.Create(DiskDrive + 'Analyse\' + IniFilename);
      IniFile.WriteString(IniShadeSetSection, IniDefaultGuide, lFilename);
    finally
      IniFile.Free;
    end;
  finally
    InStream.Free;
  end;
end;

procedure TfrmShadeAnalyzer.btnLoadClick(Sender: TObject);
begin
  OpenDialog.Title := 'Shade Guide Set to Load';
```

```
  OpenDialog.InitialDir := DiskDrive + 'Analyse\';
  OpenDialog.DefaultExt := 'SDS';
  OpenDialog.Filter := 'Shade Guides|*.SDS';
  OpenDialog.Options := [ofPathMustExist, ofFileMustExist];
  if OpenDialog.Execute then
    LoadShadeSet(OpenDialog.Filename);
end;

procedure TfrmShadeAnalyzer.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  { Closing Program - Check for unsaved Calibration Set }
  if (NewCalibration) and
     (MessageDlg('Calibration Load Not Saved. Save Now?',
                 mtConfirmation, [mbYes, mbNo], 0) = mrYes) then
    btnSaveClick(self);
end;

procedure TfrmShadeAnalyzer.FormActivate(Sender: TObject);
begin
  if Startup then
  begin
    Startup := false;
    frmSplashScreen.Show;
    Application.ProcessMessages;

{$IFDEF SLIDELOGO}
    frmSplashScreen.Timer1.Interval := 1000;
{$ELSE}
    frmSplashScreen.Timer1.Interval := 3000;
{$ENDIF}
  end;
end;

end.
```

```
unit ShadeData;

{
  The Shade reference is divided into a grid. Each area of the grid is
  analysed, and the average Red, Green and Blue values are stored. If there
  is an edge element in the sample, the valid flag is set to ignore the
  area in correlation matches.
}

interface

  uses
     Classes;

const
  GridWidth = 10;
  GridHeight = 10;

type
  PShadeColourElement = ^TShadeColourElement;
  TShadeColourElement = class(TObject)
                          Red : real48;       // Average red content
                          Green : real48;     // Average green content
                          Blue : real48;      // Average blue content
                          Valid : boolean; // Valid for comparison
                          ValidPixelPercent : real48; // 0..1 (1 = all pixels

used)
                          RedDev : real48;
                          GreenDev : real48;
                          BlueDev : real48;
                          RedMax : real48;
                          RedMin : real48;
                          GreenMax : real48;
                          GreenMin : real48;
                          BlueMax : real48;
                          BlueMin : real48;
                          constructor Create;
                          function ColourDifference (ShadeColour :

TShadeColourElement) : real48;
                          procedure StoreColour(R,G,B : real48; Percent :

real);
                          procedure AddColour(R,G,B : real48; Percent : real);
                          function ValidCell : boolean;
                          procedure SaveToStream(OutStream : TStream);
                          procedure LoadFromStream(InStream : TStream);
                        private
                       end;

  PShadeColours = ^TShadeColours;
  TShadeColours = class(TObject)
                    Name : string;            // Name of the Shade reference

etc.
                    GridColours : array[1..GridWidth, 1..GridHeight] of

TShadeColourElement;
                    function ColourDifference(ShadeColours : TShadeColours) :

real48;
                    procedure SaveToStream(OutStream : TStream);
```

```
                    procedure LoadFromStream(InStream : TStream);
                  private
                end;

 TShadeReferences = class(TObject)
                      ShadeList : TList;
                      constructor Create;
                      procedure AddSample(Sample : TShadeColours);
                      procedure Clear;
                      procedure SortList;
                      procedure ReduceList;
                      procedure SaveToStream(OutStream : TStream);
                      procedure LoadFromStream(InStream : TStream);
                    private
                  end;

implementation

uses
  SysUtils, Dialogs, Controls;

const
  ValidityLimit             = 0.95;  // 95% of pixels must be used


constructor TShadeColourElement.Create;
begin
  Red := 0;
  Green := 0;
  Blue := 0;
  ValidPixelPercent := 0;
  Valid := false;
  RedDev := 0;
  GreenDev := 0;
  BlueDev := 0;
  RedMax := 0;
  RedMin := 1000;
  GreenMax := 0;
  GreenMin := 1000;
  BlueMax := 0;
  BlueMin := 1000;
end;

procedure TShadeColourElement.StoreColour(R,G,B : real48; Percent : real);
begin
  Red := R;
  Green := G;
  Blue := B;
  ValidPixelPercent := Percent;
  Valid := (Percent >= ValidityLimit);
end;

procedure TShadeCol urElement.AddColour(R,G,B : real48; Percent : real);
begin
  if R > RedMax then RedMax := R;
  if G > GreenMax then GreenMax := G;
  if B > BlueMax then BlueMax := B;
```

```
  if R < RedMin then RedMin := R;
  if G < GreenMin then GreenMin := G;
  if B < BlueMin then BlueMin := B;
  Red := Red + R;
  Green := Green + G;
  Blue := Blue + B;
end;

function TShadeColourElement.ValidCell : boolean;
begin
  Result := Valid;
end;

function TShadeColourElement.ColourDifference(ShadeColour : TShadeColourElement)
: real48;
var
  DistanceRed : real48;
  DistanceGreen : real48;
  DistanceBlue : real48;
begin
  if (Valid) and (ShadeColour.Valid) then
  begin
    DistanceRed:=(Red - ShadeColour.Red);
    DistanceGreen:=(Green - ShadeColour.Green);
    DistanceBlue:=(Blue - ShadeColour.Blue);
    Result := sqrt(sqr(DistanceRed) +
                   sqr(DistanceGreen) +
                   sqr(DistanceBlue));

  end
  else
    Result := -1; // cannot compare if any element is invalid
end;

procedure TShadeColourElement.SaveToStream(OutStream : TStream);
begin
  OutStream.WriteBuffer(Red, SizeOf(Red));
  OutStream.WriteBuffer(Green, SizeOf(Green));
  OutStream.WriteBuffer(Blue, SizeOf(Blue));
  OutStream.WriteBuffer(Valid, SizeOf(Valid));
  OutStream.WriteBuffer(ValidPixelPercent, SizeOf(ValidPixelPercent));
  OutStream.WriteBuffer(RedDev, SizeOf(RedDev));
  OutStream.WriteBuffer(GreenDev, SizeOf(GreenDev));
  OutStream.WriteBuffer(BlueDev, SizeOf(BlueDev));
  OutStream.WriteBuffer(RedMax, SizeOf(RedMax));
  OutStream.WriteBuffer(RedMin, SizeOf(RedMin));
  OutStream.WriteBuffer(GreenMax, SizeOf(GreenMax));
  OutStream.WriteBuffer(GreenMin, SizeOf(GreenMin));
  OutStream.WriteBuffer(BlueMax, SizeOf(BlueMax));
  OutStream.WriteBuffer(BlueMin, SizeOf(BlueMin));
end;

procedure TShadeColourElement.LoadFromStream(InStream : TStream);
begin
  InStream.ReadBuffer(Red, SizeOf(Red));
  InStream.ReadBuffer(Green, SizeOf(Green));
  InStream.ReadBuffer(Blue, SizeOf(Blue));
  InStream.ReadBuffer(Valid, SizeOf(Valid));
```

```
  InStream.ReadBuffer(ValidPixelPercent, SizeOf(ValidPixelPercent));
  InStream.ReadBuffer(RedDev, SizeOf(RedDev));
  InStream.ReadBuffer(GreenDev, SizeOf(GreenDev));
  InStream.ReadBuffer(BlueDev, SizeOf(BlueDev));
  InStream.ReadBuffer(RedMax, SizeOf(RedMax));
  InStream.ReadBuffer(RedMin, SizeOf(RedMin));
  InStream.ReadBuffer(GreenMax, SizeOf(GreenMax));
  InStream.ReadBuffer(GreenMin, SizeOf(GreenMin));
  InStream.ReadBuffer(BlueMax, SizeOf(BlueMax));
  InStream.ReadBuffer(BlueMin, SizeOf(BlueMin));
end;

function TShadeColours.ColourDifference(ShadeColours : TShadeColours) : real48;
var
  DifferenceGrid : array[1..GridWidth, 1..GridHeight] of real48;
  WidthIndex : integer;
  HeightIndex : integer;
  MatchedCells : integer;
begin
  Result := 0;
  MatchedCells := 0;

  { Compare each grid positions colour }
  for WidthIndex := 1 to GridWidth do
    for HeightIndex := 1 to GridHeight do
    begin
      DifferenceGrid[WidthIndex, HeightIndex] := GridColours[WidthIndex,
HeightIndex].ColourDifference(ShadeColours.GridColours[WidthIndex,
HeightIndex]);
    end;

  {
    Calculate the Colour Difference for the whole Shade
    initially just sum the differences
    Possibly just return the standard deviation or something.
  }
  for WidthIndex := 1 to GridWidth do
    for HeightIndex := 1 to GridHeight do
      if DifferenceGrid[WidthIndex, HeightIndex] <> -1 then
      begin
        Result := Result + Sqr(DifferenceGrid[WidthIndex, HeightIndex]);
        inc(MatchedCells);
      end;
  Result := Sqrt(Result/MatchedCells);
end;

procedure TShadeColours.SaveToStream(OutStream : TStream);
var
  WidthIndex : integer;
  HeightIndex : integer;
  StringLength : integer;
begin
  StringLength := Length(Name);
  OutStream.WriteBuffer(StringLength, SizeOf(StringLength));
  OutStream.WriteBuffer(Name[1],StringLength);
  for WidthIndex := 1 to GridWidth do
    for HeightIndex := 1 to GridHeight do
```

```
      GridColours[WidthIndex, HeightIndex].SaveToStream(OutStream);
end;

procedure TShadeColours.LoadFromStream(InStream : TStream);
var
  WidthIndex : integer;
  HeightIndex : integer;
  StringLength : integer;
begin
  InStream.ReadBuffer(StringLength, SizeOf(StringLength));
  SetLength(Name, StringLength);
  InStream.ReadBuffer(Name[1],StringLength);
  for WidthIndex := 1 to GridWidth do
    for HeightIndex := 1 to GridHeight do
    begin
      GridColours[WidthIndex, HeightIndex] := TShadeColourElement.Create;
      GridColours[WidthIndex, HeightIndex].LoadFromStream(InStream);
    end;
end;

procedure TShadeReferences.AddSample(Sample : TShadeColours);
begin
  ShadeList.Add(Sample);
end;

procedure TShadeReferences.Clear;
begin
  ShadeList.Clear;
end;

constructor TShadeReferences.Create;
begin
  ShadeList := TList.Create;
end;

function SortCompare(Item1, Item2: pointer): Integer;
begin
  if TShadeColours(Item1).Name < TShadeColours(Item2).Name then
    Result := -1
  else if TShadeColours(Item1).Name > TShadeColours(Item2).Name then
    Result := 1
  else
    Result := 0;
end;

procedure TShadeReferences.SortList;
begin
  ShadeList.Sort(SortCompare);
end;

procedure TShadeReferences.ReduceList;
var
  AverageShadeColours : TShadeColours;
  AveragedShades : TList;
  CurrentShade : TShadeColours;
  ShadeIndex : integer;
  Row, Col : integer;
```

```
  AverageCount : array[1..GridWidth, 1..GridHeight] of integer;
begin
  { For each individually Named shade, average all values into one ShadeColours
}

//  New(AverageShadeColours);

  AveragedShades := TList.Create;

  //New(CurrentShade);
  CurrentShade := TShadeColours.Create;
  CurrentShade.Name := '';

  for ShadeIndex := 1 to ShadeList.Count do
  begin
    if CurrentShade.Name <> TShadeColours(ShadeList.Items[ShadeIndex-1]).Name
then
    begin
      if ShadeIndex <> 1 then
      begin
        { Save the last shade and start a new one }
        for Col := 1 to GridWidth do
          for Row := 1 to GridHeight do
            if AverageCount[Col, Row] <> 0 then
              with AverageShadeColours.GridColours[Col, Row] do
              begin
                Red := Red / AverageCount[Col, Row];
                Green := Green / AverageCount[Col, Row];
                Blue := Blue / AverageCount[Col, Row];
                ValidPixelPercent := ValidPixelPercent / AverageCount[Col, Row];
               end;
         AveragedShades.Add(AverageShadeColours);
       end;

       {This is a new Shade}
       AverageShadeColours := TShadeColours.Create;
       for Col := 1 to GridWidth do
         for Row := 1 to GridHeight do
         begin
           AverageShadeColours.GridColours[Col,Row] :=
  TShadeColourElement.Create;
           AverageCount[Col, Row] := 0;
         end;
       AverageShadeColours.Name:= TShadeColours(ShadeList.Items[ShadeIndex-
  1]).Name;
     end;

     CurrentShade := ShadeList.Items[ShadeIndex-1];

     for Col := 1 to GridWidth do
       for Row := 1 to GridHeight do
       begin
         with AverageShadeColours.GridColours[Col, Row] do
         begin
           if CurrentShade.GridColours[Col, Row].Valid then
           begin
             Valid := true;
```

```
            AddColour(CurrentShade.GridColours[Col, Row].Red,
                      CurrentShade.GridColours[Col, Row].Green,
                      CurrentShade.GridColours[Col, Row].Blue,
                      CurrentShade.GridColours[Col, Row].ValidPixelPercent);
//            ValidPixelPercent := ValidPixelPercent +
CurrentShade.GridColours[Col, Row].ValidPixelPercent;
//            Red := Red + CurrentShade.GridColours[Col, Row].Red;
//            Green := Green + CurrentShade.GridColours[Col, Row].Green;
//            Blue := Blue + CurrentShade.GridColours[Col, Row].Blue;
            inc(AverageCount[Col, Row]);
          end;
        end;
      end;
  end;

  { Save the last shade }
  for Col := 1 to GridWidth do
    for Row := 1 to GridHeight do
      if AverageCount[Col, Row] <> 0 then
        with AverageShadeColours.GridColours[Col, Row] do
        begin
          Red := Red / AverageCount[Col, Row];
          Green := Green / AverageCount[Col, Row];
          Blue := Blue / AverageCount[Col, Row];
          ValidPixelPercent := ValidPixelPercent / AverageCount[Col, Row];
        end;

  AveragedShades.Add(AverageShadeColours);

  ShadeList.Free;
  ShadeList := AveragedShades;
end;

procedure TShadeReferences.SaveToStream(OutStream : TStream);
var
  ShadeIndex : integer;
  CurrentShade : TShadeColours;
begin
  ShadeIndex := ShadeList.Count;  // First write the number of Shade in the set
  OutStream.WriteBuffer(ShadeIndex, SizeOf(ShadeIndex));
  for ShadeIndex := 0 to ShadeList.Count - 1 do
  begin
    CurrentShade := ShadeList.Items[ShadeIndex];
    CurrentShade.SaveToStream(OutStream);
  end;
end;

procedure TShadeReferences.LoadFromStream(InStream : TStream);
var
  NumberOfShades : integer;
  ShadeIndex : integer;
  CurrentShade : TShadeColours;
begin
  InStream.ReadBuffer(NumberOfShades, SizeOf(NumberOfShades));
  for ShadeIndex := 0 to NumberOfShades - 1 do
  begin
    CurrentShade := TShadeColours.Create;
```

```
    CurrentShade.LoadFromStream(InStream);
    AddSample(CurrentShade);
  end;
end;

end.
```

```
unit SplashScreen;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, TMultiP;

type
  TSplashState = (splCenter, splMoving, splDone);

  TfrmSplashScreen = class(TForm)
    Timer1: TTimer;
    pnlLogo: TPanel;
    Image1: TImage;
    procedure Timer1Timer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
  private
    { Private declarations }
    SplashState : TSplashState;
    StartPosition : TPoint;
    StartSize : TPoint;
    VirticalStep : integer;
    HorizontalStep : integer;
    WidthStep : integer;
    HeightStep : integer;
    CanPaint : boolean;
  public
    { Public declarations }
  end;

var
  frmSplashScreen: TfrmSplashScreen;

implementation

{$R *.DFM}

const
  FinishPosition : TPoint = (x:580; y:700);
  FinishSize     : TPoint = (x:430; y:50);
  Duration                = 2000; // ms
  MoveInterval            = 50;   // ms

procedure TfrmSplashScreen.Timer1Timer(Sender: TObject);
begin
{$IFDEF SLIDELOGO}
  if abs(FinishPosition.x - Left) < abs(HorizontalStep) then
    HorizontalStep := HorizontalStep div abs(HorizontalStep);
  if abs(FinishPosition.Y - Top) < abs(VirticalStep) then
    VirticalStep := VirticalStep div abs(VirticalStep);

  if abs(FinishSize.x - Width) < abs(WidthStep) then
     WidthStep := WidthStep div abs(WidthStep);
  if abs(FinishSize.Y - Height) < abs(HeightStep) then
    HeightStep := HeightStep div abs(HeightStep);
```

```
CanPaint := false;

case SplashState of
  splCenter:
    begin
      Timer1.Interval := MoveInterval;
      SplashState := splMoving;
    end;
  splMoving:
    begin
      if (Top = FinishPosition.y) and (Left = FinishPosition.x) and
         (Height = FinishSize.y) and (Width = FinishSize.x) then
        SplashState := splDone
      else
      begin
        if Top <> FinishPosition.y then
          Top := Top + VirticalStep;
        if Left <> FinishPosition.x then
          Left := Left + HorizontalStep;
        if Height <> FinishSize.y then
          Height := Height + HeightStep;
        if Width <> FinishSize.x then
          Width := Width + WidthStep;
      end;
    end;
  splDone:
    begin
      Timer1.Enabled := false;
    end;
end;
CanPaint := true;
{$ELSE}
Height := FinishSize.y;
Width := FinishSize.x;
Top := FinishPosition.y;
Left := FinishPosition.x;
pnlLogo.BevelWidth := 2;
{$ENDIF}
end;

procedure TfrmSplashScreen.FormCreate(Sender: TObject);
begin
  SplashState := splCenter;
  StartPosition.x := Left;
  StartPosition.y := Top;
  StartSize.x := Width;
  StartSize.y := Height;
//VirticalStep := (FinishPosition.y - StartPosition.y) div (Duration div
MoveInterval);
//HorizontalStep := (FinishPosition.x - StartPosition.x) div (Duration div
MoveInterval);
//HeightStep := (FinishSize.y - StartSize.y) div (Duration div MoveInterval);
//WidthStep := (FinishSize.x - StartSize.x) div (Duration div MoveInterval);
  VirticalStep := 1;
  HorizontalStep := 1;
  HeightStep := -1;
```

```
  WidthStep := -1;
  CanPaint := true;
end;

procedure TfrmSplashScreen.FormPaint(Sender: TObject);
begin
  if CanPaint then
    Image1.Repaint;
end;

end.
```

```
unit ToothObject;

interface

uses
  Windows, SysUtils, Classes, Graphics,
  ShadeData;

type
  TTooth = class(TObject)
      private
         ReferenceInitialised : boolean;
         function CalculateTestArea(Row, Col : integer;
                                    Area : TRect;
                                    NoRows, NoCols : integer) : TRect;


      public
         Red : real;
         Green : real;
         Blue : real;
         Hue : real;
         Saturation : real;
         Intensity : real;
         RefRed : real;
         RefGreen : real;
         RefBlue : real;
         RefHue : real;
         RefSaturation : real;
         RefIntensity : real;
         ToothBitmap : TBitmap;
         ToothBitmapMask : TBitmap;
         constructor Create;
         procedure Free;
         procedure LoadBitmapFromFile(Filename : String);
         procedure RemoveReflection(TestArea : TRect);
         procedure RemoveMask(TestArea : TRect);
         function FillSearchSampleLimits(StartPoint : TPoint) : TRect;
         procedure Analyse(Row, Col : integer;
                           Area : TRect; NoRows, NoCols : integer;
                           var R, G, B : real;
                           var PixelPercentage : real);
         function AnalyseGrid(Area : TRect;NoRows, NoCols : integer;
                              DeltaRed, DeltaGreen, DeltaBlue : real) :
TShadeColours;
         procedure CalculateHSI(R, G, B : real; var Hue, Sat, Int : real);
    end;


implementation

uses
  Dialogs, Math;

const
  RedMask   : longint      = $000000FF;
  GreenMask : longint      = $0000FF00;
```

```
  BlueMask   : longint      = $00FF0000;

  BoundryIntensity          = 0.58; // sum of RGB
  ReflectionIntensity       = 1.86;

type
  TDirection = (Up, Down, Left, Right);

constructor TTooth.Create;
begin
    ToothBitmap := TBitmap.Create;
    ToothBitmap.Width := 640;
    ToothBitmap.Height := 480;
    ToothBitmapMask := TBitmap.Create;
    ToothBitmapMask.Width := 640;
    ToothBitmapMask.Height := 480;
    ReferenceInitialised := false;
end;

procedure TTooth.Free;
begin
  ToothBitmap.Free;
  ToothBitmapMask.Free;
  Inherited Free;
end;

function TTooth.FillSearchSampleLimits(StartPoint : TPoint) : TRect;
  procedure FillSearch(StartpointX, StartpointY : integer);
  var
    Red, Green, Blue : integer;
    x, y : integer;
  begin
    if StartPointX > Result.Right then
      Result.Right := StartpointX;
    if StartpointX < Result.Left then
      Result.Left := StartpointX;
    if StartpointY > Result.Bottom then
      Result.Bottom := StartpointY;
    if StartpointY < Result.Top then
      Result.Top := StartpointY;

    for x := -1 to 1 do
      for y := -1 to 1 do
        if (StartpointX+x < 640) and (StartpointX+x > 0) and
           (StartpointY+y < 480) and (StartpointY+y > 0) and
           (ToothBitmapMask.Canvas.Pixels[StartpointX+x, StartpointY+y] <> 0)
then
        begin
          Red := ToothBitmap.Canvas.Pixels[StartpointX+x, StartpointY+y] and
RedMask;
          Green := (ToothBitmap.Canvas.Pixels[StartpointX+x, StartpointY+y] and
GreenMask) shr 8;
          Blue := (ToothBitmap.Canvas.Pixels[StartpointX+x, StartpointY+y] and
BlueMask) shr 16;
          if ((Red + Green + Blue)/255) > BoundryIntensity then
          begin
            ToothBitmapMask.Canvas.Pixels[StartpointX+x, StartpointY+y] := 0;
```

```
            FillSearch(StartpointX+x,StartpointY+y);
          end
        end;
  end;
begin
  Result := Rect(640, 480, 0, 0);
  FillSearch(StartPoint.X, StartPoint.Y);
end;

function TTooth.CalculateTestArea(Row, Col : integer;
                                  Area : TRect;
                                  NoRows, NoCols : integer) : TRect;

var
  TestArea : TRect;
  ColSpacing : real;
  RowSpacing : real;
begin
  with Area do
  begin
    ColSpacing := (Right - Left) / NoCols;
    RowSpacing := (Bottom - Top) / NoRows;
    TestArea.Top := Top + Trunc(RowSpacing * Row);
    TestArea.Bottom := Top + Trunc(RowSpacing * (Row + 1));
    TestArea.Left := Left + Trunc(ColSpacing * Col);
    TestArea.Right := Left + Trunc(ColSpacing * (Col + 1));
  end;

  CalculateTestArea := TestArea;
end; // CalculateTestArea

procedure TTooth.RemoveReflection(TestArea : TRect);
var
   i : integer;
   j : integer;
   Red, Green, Blue : integer;
begin
    with TestArea do
    for i := Left to Right do
        for j := Top to Bottom do
        begin
            if (ToothBitMapMask.Canvas.Pixels[i,j] <> 0) then
              ToothBitMap.Canvas.Pixels[i,j] := 0
            else
            begin
              Red := ToothBitMap.Canvas.Pixels[i,j] and RedMask;
              Green := (ToothBitMap.Canvas.Pixels[i,j] and GreenMask) shr 8;
              Blue := (ToothBitMap.Canvas.Pixels[i,j] and BlueMask) shr 16;

              if ((Red + Green + Blue) / 255 > ReflectionIntensity) then
                ToothBitMap.Canvas.Pixels[i,j] := 0;
            end;
        end;


end; // RemoveReflection

procedure TTooth.RemoveMask(TestArea : TRect);
```

```
var
   i : integer;
   j : integer;
begin
    with TestArea do
    for i := Left to Right do
        for j := Top to Bottom do
            if (ToothBitMapMask.Canvas.Pixels[i,j] <> 0) then
                ToothBitMap.Canvas.Pixels[i,j] := 0;


end; // RemoveReflection

procedure TTooth.Analyse(Row, Col : integer;
                         Area : TRect; NoRows, NoCols : integer;
                         var R, G, B : real;
                         var PixelPercentage : real);

var
  TestArea : TRect;
   PixelCount : longint;

   i : integer;
   j : integer;
   RedTotal : longint;
   GreenTotal : longint;
   BlueTotal : longint;
   Red, Green, Blue : integer;
begin
    TestArea := CalculateTestArea(Row,Col,Area,NoRows,NoCols);
    with TestArea do
      PixelCount := (Right - Left + 1) * (Bottom - Top + 1);

    // Now average ignoring reflections and blemishes
    RedTotal := 0;
    GreenTotal := 0;
    BlueTotal := 0;

    with TestArea do
    for i := Left to Right do
        for j := Top to Bottom do
        begin
            if ToothBitMap.Canvas.Pixels[i,j] <> 0 then
            begin
              Red := ToothBitMap.Canvas.Pixels[i,j] and RedMask;
              Green := (ToothBitMap.Canvas.Pixels[i,j] and GreenMask) shr 8;
              Blue := (ToothBitMap.Canvas.Pixels[i,j] and BlueMask) shr 16;

              RedTotal := RedTotal + Red;
              GreenTotal := GreenTotal + Green;
              BlueTotal := BlueTotal + Blue;
            end
            else
            begin
                PixelCount := PixelCount - 1; // Ignored this pixel
            end;
        end;
```

```
    // Normalised RGB
    if PixelCount > 0 then
    begin
      R := RedTotal / PixelCount / 255;
      G := GreenTotal / PixelCount / 255;
      B := BlueTotal / PixelCount / 255;
      with TestArea do
        PixelPercentage := PixelCount / ((Bottom-Top+1)*(Right - Left+1));
    end
    else
    begin
      R := 0;
      G := 0;
      B := 0;
      PixelPercentage := 0;
    end;
end; // Analyse

function TTooth.AnalyseGrid(Area : TRect;
                            NoRows, NoCols : integer;
                            DeltaRed, DeltaGreen, DeltaBlue : real) :

TShadeColours;
var
  Row, Col : integer;
  Red, Green, Blue, PixelPercent : real;
begin
  Result := TShadeColours.Create;

  for Row := 0 to NoRows - 1 do
    for Col := 0 to NoCols - 1 do
    begin
      Analyse(Row, Col, Area, NoRows, NoCols,
                    Red, Green, Blue, PixelPercent);
      Result.GridColours[Col+1, Row+1] := TShadeColourElement.Create;
      Result.GridColours[Col+1, Row+1].StoreColour(Red + DeltaRed,
                                                   Green + DeltaGreen,
                                                   Blue + DeltaBlue,
                                                   PixelPercent);

    end;
end;

procedure TTooth.CalculateHSI(R, G, B : real; var Hue, Sat, Int : real);

  function Minimum (v1, v2, v3 : real) : real;
  begin
    if (v1 <= v2) and (v1 <= v3) then
        minimum := v1
    else
        if (v2 <= v1) and (v2 <= v3) then
            Minimum := v2
        else
            Minimum := v3;
  end;

  function Maximum (v1, v2, v3 : r al) : real;
  begin
    if (v1 >= v2) and (v1 >= v3) then
```

```
        Maximum := v1
    else
        if (v2 >= v1) and (v2 >= v3) then
            Maximum := v2
        else
                Maximum := v3;
end;

begin
    // Calculation using Gonzalez and Woods
    Int := (R + G + B) / 3;

    if Int > 0 then
    begin
      Sat := 1 - (3 / (R + G + B)) * Minimum(R,G,B);
      Hue := arccos((((R-G)+(R-B))/2)/sqrt(sqr(R-G)+((R-B)*(G-B)))) /(2*pi);
      if (B / Int) > (G / Int) then
         Hue := 1 - Hue;
    end
    else
    begin
      Sat := 0;
      Hue := 0;
    end;
end; // CalculateHSI

procedure TTooth.LoadBitmapFromFile(Filename : String);
begin
  ToothBitmap.LoadFromFile(Filename);
  ToothBitmap.Dormant;
  ToothBitmapMask.Assign(ToothBitmap);
end; // LoadBitmapFromFile


end.
```